HT21

Projektidéer

INTE HT2021

HENRIK BERGSTRÖM

Introduktion

Detta dokument innehåller förslag på uppgifter för projektet på INTE. Gruppen väljer själv vilket projekt den vill arbeta med, och vem som gör vilken del. Dessa val behöver inte anmälas i förväg, utan redovisas på det första redovisningstillfället. Om inget av idéerna passar står det gruppen fritt att komma med ett eget förslag. Sådana förslag måste dock godkännas av kursledningen i förväg.

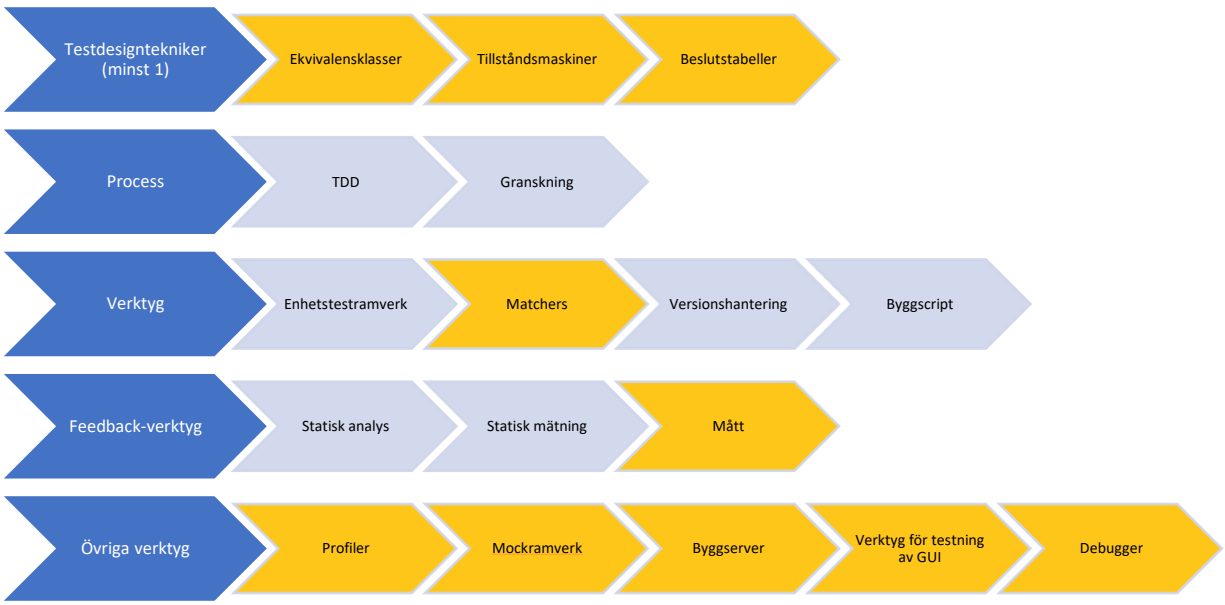
För att ett förslag ska godkännas måste det vara av lämplig omfattning och komplexitet, samt innehålla både gemensamma och individuella delar. De individuella delarna måste också vara av sådan omfattning och komplexitet att alla gruppmedlemmar får möjlighet att applicera det som tagits upp under kursen.

Det finns inga explicita gränser för vad som är lämplig omfattning och komplexitet, utan det avgörs från fall till fall. Om allt som görs är att utveckla klass efter klass med enbart data och get och set-metoder spelar det ingen roll hur många klasser som tas fram. Det kommer inte att vara intressant att testa. Tänk också på att inte lägga all funktionalitet i samma klass, det är inte en objektorienterad lösning. Klasserna ska kombinera data och funktionalitet, och samarbeta för att lösa olika uppgifter som inte bara kräver enkla variabler och villkor. Det är också viktigt att de individuella delarna påverkar varandra så att versionshanteringssystemet kommer till nytta.

Oavsett om gruppen väljer en av de föreslagna projektidéerna eller kommer med sitt eget förslag är det viktigt att tänka på att syftet är att få fram något intressant för en utvecklare att testa, inte att utveckla ett klassbibliotek eller system. Det finns inga krav att allt blir klart, eller att alla varianter gås igenom.

Ickefunktionella krav på uppgiften

Förutom de funktionella kraven på uppgiften, som ni i stor utsträckning bestämmer själva, så finns det också ett antal ickefunktionella krav på den. De flesta av dessa handlar om att ni ska använda specifika verktyg och tekniker, och sammanfattas i figuren nedan. De ljust blå-gråa är obligatoriska att använda. För de gula gäller att gruppen väljer (minst) fyra av dem, varav minst en av testdesignteknikerna. De ickefunktionella kraven diskuteras på projektintroduktionen efter tentan.



Dependency injection

Dependency injection är en vanlig teknik för att öka testbarheten hos något. Denna teknik kombineras ofta ihop med test doubles (tex stubbar eller mock-objekt), och är något som ni bör testa på att använda er av. Det är en mycket vanlig teknik i verkliga projekt, och kan med fördel användas till exempel när man vill undvika att testa med riktiga filer, internetuppkopplingar eller databaser. Flera av uppgifterna har förslag på var man kan tillämpa detta.

Designmönster

På flera av uppgiftsförslagen finns det också tips om att olika designmönster kan vara lämpliga att använda. Det finns inget krav på att ni verkligen implementerar funktionerna med hjälp av dessa designmönster, men det kan bidra till att göra uppgiften mer realistisk och mer intressant ur testsynvinkel.

De vanligaste referenserna är till Decorator[1] och Visitor[2], men även Observer[3] (Listener) är av intresse på många ställen. En teknik som också kan vara relevant om ni har klasshierarkier är Double dispatch[4].

Slutligen: undvik Singelton[5]. De är svåra att testa, och kan vara mycket svåra att få bort om man kommer på att det skulle behövas flera.

[1] https://en.wikipedia.org/wiki/Decorator_pattern
[2] https://en.wikipedia.org/wiki/Visitor_pattern
[3] https://en.wikipedia.org/wiki/Observer_pattern
[4] https://en.wikipedia.org/wiki/Double_dispatch
[5] https://en.wikipedia.org/wiki/Singleton_pattern

Rougelike

Denna uppgift är lämplig för grupper på tre till fem personer, och har varit den rekommenderade uppgiften på kursen de senaste åren. Uppgiften fungerar väl, men kan av vissa uppfattas ha väl stora frihetsgrader. Det kan helt enkelt vara svårt att välja när det finns så många olika saker att välja mellan.

Uppgiften är att utveckla ett klassbibliotek som skulle kunna användas för att utveckla ett så kallat ”rougelike”- spel. Det finns mycket information om dessa på nätet. Ett bra ställe att börja är Wikipedia: https://en.wikipedia.org/wiki/Roguelike, men i grund och botten handlar det om en ensam spelare som utforskar en slumpad karta, vanligen en grotta full av monster, skatter och magisk utrustning som förändrar spelarens egenskaper. Det underlättar om man inom gruppen har erfarenhet av speltypen, men det är inget krav, och kan i vissa fall leda till att man låser sig i hur det ser ut i existerande spel.

Av testdesignteknikerna är tillståndsmaskiner den enklaste att applicera i denna typ av projekt, till exempel för NPC:er, uppdrag eller magi. Det är fullt möjligt att applicera beslutstabeller eller ekvivalensklasser också, men det kräver ofta lite mer planering för att hitta något intressant ställe.

Vill man använda en profiler behöver man något som drar resurser. En variant kan vara en stor karta med många npc:er som rör sig samtidigt.

Ett starkt tips är att inte utveckla något grafiskt användargränssnitt, eller någon spelmotor som faktiskt spelar spelet. Om gruppen vill testa på att använda verktyg för testning av grafiska användarinterface är det lämpligt att man gör det för en mycket liten del, till exempel en dialog för att skapa en spelare.

Förslag på uppdelning

Här nedan finns förslag på uppgifter som lämpar sig för en deltagare i projektet. Vissa av dem går in i varandra, och vissa, som magi, kan göras hur stora som helst och skulle kunna delas upp på flera personer beroende på vad exakt man väljer att göra.

Spelaren

Spelar-karaktären med ett par olika numeriska attribut som kommer att påverka olika saker i spelet. Lägg inte till för många attribut till att börja med, kom ihåg att mer inte nödvändigtvis är bättre. Deltagarna i projektet kan senare lägga till ytterligare attribut när det blir nödvändigt.

Denna del kan vara lämplig att utveckla åtminstone grunderna av tillsammans då den antagligen kommer att påverka alla deltagarna i projektet.

Om spelaren istället ska utvecklas av en deltagare så behövs det mer funktionalitet. Typiska saker som kan finns är till exempel erfarenhetspoäng (XP) och nivåer som kan gå såväl upp som ner, raser eller yrken som påverkar attributen och hur snabbt man går upp och ner i nivå, och olika sekundära egenskaper som spelaren kan lära sig när hen går upp i nivå, och glömma bort om nivån minskar. Ett tips är att titta på decorator-mönstret.

Det finns också delar av de övriga förslagen nedan som berör spelaren.

Karta och terräng

Karta med olika typer av terräng som det kostar olika mycket att ta sig fram över, eller bara kan kommas åt av vissa typer av karaktärer, tex sådana som kan simma eller flyga.

Terrängen kanske kan förändra sig över tid, till exempel snö som smälter, radioaktivitet som sprider sig, eller årstider som växlar.

Hur stora delar av kartan är synlig för spelaren eller en npc? Bara den delen som är utforskad, eller en bit runt omkring, och i så fall hur långt? Kan man se igenom en kulle?

Det finns algoritmer för att slumpmässigt ta fram kartor av olika typer som skulle kunna vara intressanta att testa.[6]

Ett annat alternativ om man inte vill generera kartor är att läsa in dem från fil istället och då tolka ett filformat. Detta är ett bra ställe att experimentera med dependency injection eftersom man gärna undviker att enhetstester läser från filsystemet annat än i undantagsfall eftersom det är långsamt och lätt ställer till problem med sökvägar när andra deltagare i projektet ska köra testerna.

Magi-system

Detta område är så stort att flera personer lätt kan arbeta på det parallellt. Det viktiga här är att inte göra det alltför enkelt för sig. Om allt man har är en variabel som säger att spelaren gör magisk skada är det garanterat för litet och för enkelt.

Magi kan påverkas av saker som nivå, ras, yrke, vilka formler man lärt sig, eller glömt, vilka ingredienser man har, tiden på dygnet, månens faser, vad man försöker göra med det, etc. Det kan påverka spelaren eller en NPC direkt eller indirekt, permanent eller temporärt, avslöja andra delar av kartan, ge ledtrådar till uppdrag, förbättra utrustning, etc.

Även här kan decorator-mönstret vara av intresse för vissa typer av magi, liksom Visitor och Double Dispatch.

Utrustning

Det finns ofta olika typer av utrustning som man kan hitta, köpa, eller få som belöning. Beroende på hur komplext systemet är kan detta vara alltifrån en liten del av spelarkaraktären till en stor del av systemet.

Utrustningen kan vara av olika typ, bara användas av vissa raser eller yrken, påverkas av magi, läggas till och tas bort ur en begränsad samling av saker spelaren kan bära med sig, sättas på och tas av, och då rimligen bara en av varje typ förutom ringar som man kan ha åtta eller tio av, etc. En spelare med yrket eller egenskapen ficktjuv kan stjäla mindre saker från en annan spelare eller NPC till exempel.

NPC och story

Vissa NPC:er kan ge spelaren uppdrag av olika typ, tex prata med någon annan NPC, ta sig till en viss plats, hämta något speciellt, besegra en viss fiende, svara rätt på en gåta, etc. När spelaren genomfört ett uppdrag händer något i spelet, tex spelaren får något, en ny del av kartan låses upp, en annan NPC vill nu prata med spelaren, etc.

Uppdrag av lite mer komplex modell styrs ofta av tillståndsmaskiner.

Ras och yrkesfärdigheter

Om man har egenskaper som ras eller yrke kan dessa variera stort i vad de kan göra. Den ovan nämnda ficktjuven är helt annorlunda än smeden, som i sin tur är annorlunda än prästen, köpmannen, trollkarlen, etc. Dessa olikheter kan appliceras på såväl spelare som NPC:er.

Vissa spel ger spelaren möjlighet att lära sig nya yrken, eller byta under spelets gång.

AI

Hur rör sig NPC:erna på kartan? Hur samarbetar de? Slåss? Genomför sina dagliga sysslor?

Detta styrs ofta av script eller tillståndsmaskiner.

[6] Sökord: procedural map generation

Kassasystem

Denna uppgift är lämplig för grupper på tre till fem personer, den är betydligt renare än den föregående, men kan upplevas som lite tråkigare eftersom den är betydligt mer styrd av hur denna typ av system fungerar i verkligheten.

Uppgiften går ut på att implementera klasser som skulle kunna användas i ett kassasystem med varor, kvitton, rabatter, kunder, etc.

Samtliga testdesigntekniker går att applicera i denna typ av projekt. Vilka som är lämpligast beror på exakt vad man gör.

En profiler kan vara intressant om man vill studera hur någon del av systemet klarar stora mängder data, eller vid generering av rapporter som ställer samman större datamängder.

Till skillnad från rougelike-förslaget kan det här eventuellt vara intressant att utveckla en liten del av ett användarinterface, eller åtminstone något som kopplar ihop de olika delarna, dock absolut inte hela kassasystemet.

Förslag på uppdelning

Nedanstående lista innehåller förslag på funktioner som kan implementeras. Några av dessa, som pengar och varor är absolut nödvändiga, och är lämpliga att utveckla tillsammans för att komma igång i projektet. Därefter kan varje person välja ett antal funktioner och gruppen jobba mer distribuerat. Det går inte att säga exakt hur många funktioner man ska ta per person eftersom detta beror på deras komplexitet. Tänkt dock på att varje person måste ha tillräckligt mycket intressant funktionalitet, och också möjlighet att interagera med andra deltagares kod. Det är också viktigt att arbeta objektorienterat. Rabatter till exempel ska inte räknas ut av kvittoklassen, även om den säkert ansvarar för att samla ihop dem.

Pengar

I princip det som gjordes när vi gick igenom TDD. En klass som representerar pengar.

Valutor

En möjlig utökning av ovanstående så att systemet stödjer flera valutor. Kan ge intressanta effekter om man tänker på växelkurser, och vad som ska hända om kunden inte betalar med ett exakt belopp. I vilken valuta ska kunden då få växel?

Varor

Som ett minimum ett namn och ett pris. Antagligen också förmågan att presentera sig själv på något lämpligt sätt.

Varugrupper

Samlingar av varor med något gemensamt, ofta använt i samband med rabatter, eller för att generera statistik. Man bör kunna lägga till och ta bort varor ur gruppen, samt iterera över den[7].

Momskategorier

Olika varutyper har olika momssats som behöver räknas ut och redovisas på kvitton och eventuella relevanta rapporter.

Pant

Vissa varor, till exempel många dryckesförpackningar, har en separat pant som läggs på utöver priset, och som man (rimligen) inte betalar moms eller får rabatt på.

[7] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/Iterable.html

Åldersgränser

Vissa varor, som alkohol och tobak, får bara köpas av personer över en viss ålder. Detta är bara intressant om man implementerar själva kassan, annars blir det bara ett fält i varan eller varukategorin.

Leverantörer

Egentligen samma sak som varugrupper, men kopplade till en viss leverantör.

Kvitton

En samling med varor som automatiskt kan räkna ut pris. Funktioner man kan tänka sig är lägga till och ta bort varor, gruppera varor, sortera dem i bokstavsordning, samt naturligtvis skriva ut kvittot. Tänk på att samma vara kan läggas till och tas bort vid olika tillfällen.

Java har ett ganska enkelt interface för att skriva ut på en skrivare[8] som kan implementeras, och med fördel kan testas med hjälp av, till exempel, mock-objekt. Kvitton har normalt också datum och tid som kan testas med hjälp av test doubles.

Oavsett om man genererar kvittot i textform, som en bild, eller skriver ut det på riktigt, så måste man tänka på vad som händer om namnet på varan är för långt, och på att talen bör komma i prydliga kolumner. Formateringssträngar och fonter med fast bredd är användbara här. Fonter kan också ställa till problem eftersom dessa kan skilja sig åt mellan datorer och operativsystem.

Många av de andra funktionerna kan påverka kvittot, till exempel personen som sitter i kassan, kundens eventuella medlemskap, momssatser, etc.

Kassan

En driver för många av de andra funktionerna. Exakt vad denna ska innehålla beror på vad man väljer att implementera, och det finns inga krav på att man överhuvudtaget gör denna.

Växel

Om kunden betalar kontakt ska systemet lämna växel med minsta antal sedlar och mynt. Idag hanteras detta ofta av automtiska system, vad händer med ett sådant om det inte finns tillräckligt mycket av en valör? I vissa fall kan man ge fler av en lägre valör, i andra inte.

EAN/GTIN[9]

En naturlig utökning av varuklassen skulle vara en klass som representerar streckkodens GTIN (Global Trade Item Number). Validering av kontrollsiffran är en funktion här.

Generering av den associerade streckkoden skulle kunna vara en intressant utökning, och kan göras antingen till en bild eller till en skrivare.

Scanning

Scanning av varorna är det vanliga sättet att registrera köp. Ni har inte tillgång till en scanner, så att försöka lägga till en sådan funktionalitet skulle kunna vara ett intressant exempel på hur man testar interaktion med något som ännu inte är implementerat med hjälp av iInterface och test doubles.

Tänk på att scanning inte alltid lyckas.

Våg

En variant av ovanstående är att kunna väga varor. Detta skulle också få in en ny typ av varor där priset är per viktenhet istället för per styck.

Tänk på att olika varor använder olika viktenheter. Saffran mäts ofta i gram, godis i hekto, frukt och grönsaker i kilo och grus i ton.

[8] https://docs.oracle.com/javase/tutorial/2d/printing/index.html

[9] https://sv.wikipedia.org/wiki/European_Article_Number

Enkla rabatter

Enkla rabatter är sådana som ger ett visst belopp eller en viss procentsats i rabatt. De kan vara kopplade till hela köpet, en viss vara eller en varugrupp. Nästan alltid har de en giltighetstid som kan testas med hjälp av dependency injection.

Ett designmönster som kan vara intressant här är Decorator.

Mer komplicerade rabatter

Mer komplicerade rabatter är till exempel tag 3 betala för 2, rabatter som bara gäller max 2 köp per kund, rabatter som kräver att kunden handlat över ett visst belopp med eller utan den rabatterade varan, eller att det är en speciell tid på dagen och kanske bara gäller vissa kunder så som pensionärsrabatter dagtid vissa dagar, eller happy hour.

Bara en rabatt

Oavsett vilka rabatter man har brukar det i de flesta fall vara så att max en rabatt kan tillämpas på en viss vara, och då är det viktigt att det är den som ger högst rabatt som används.

I vissa fall kan detta bli ganska komplicerat, speciellt om det finns vissa varor som ingår i flera rabatterbjudanden, eller om det finns rabatter som gäller hela köpet samtidigt som det finns rabatter på vissa varor. I det senare fallet kan det i vissa fall vara bättre att räkna bort den rabatterade varan om rabatten på hela köpet är högre.

Lagersaldo

En naturlig utökning av ett kassasystem är att också hålla reda på lagersaldot för varorna i butiken, och automatiskt uppdatera detta när någonting köps.

Kunder

Namn, adress, personnummer, telefonnummer, e-postadress. Ju mer information vi har om kunderna desto bättre. Självklart vill vi också hålla reda på vad varje kund handlar så att vi kan generera statistik och riktad reklam.

Ett tips är dock att inte försöka få in all information ni kan komma på i klassen. Det är lätt att det blir för stort fokus på kvantitet istället för kvalitet. Välj ut delar som blir intressanta ur testsynvinkel, eller som är nödvändiga för andra funktioner.

Medlemskap

Många affärskedjor erbjuder medlemskap till sina kunder. Dessa ger ofta någon typ av fördel för kunden, samtidigt som de tillåter affären att samla in betydligt mer information om kunden och dennes köp.

Medlemskap kan också vara tidsbegränsade, kosta pengar, ha en minimiålder, etc.

Återbäring

En fördel som kan finnas med ett medlemskap är direkt återbäring, där medlemmen får tillbaka en del av köpesumman. Detta kan ske när kunden handlat för tillräckligt mycket inom en viss tid, eller med viss periodicitet. Självklart är det viktigt att köp man redan fått återbäring på inte räknas med nästa gång återbäringen ska betalas ut.

Ibland finns det också olika nivåer på återbäring beroende på hur mycket man har handlat för eller vilken typ av medlemskap man har.

Vissa kategorier av varor kan vara uteslutna ur återbäringsprogram, till exempel lotter och presentkort.

Poäng

En variant på återbäring är att man istället samlar poäng som sen kan bytas in mot varor eller presentkort.

En typ av erbjudande som ibland förekommer är att medlemmar får dubbla poäng under en viss period, eller på vissa varor eller kategorier.

Bonuscheckar

En sak som ofta kan köpas för poäng är bonuscheckar som sen kan användas för att helt eller delvis betala ett köp. Dessa räknas normalt inte som rabatter, så man kan betala ett köp med flera bonuscheckar, och eventuella

rabatter ska appliceras. En möjlig begränsning är att man inte får ut något av värdet för en bonuscheck i kontanter, så att de alltså inte ger någon växel om man betalar för mycket.

Presentkort

En annan form av alternativt betalningsmedel är presentkort. Dessa är varor, men räknas aldrig med i rabatter eller bonussystem, och måste ibland aktiveras för att vara giltiga. De har normalt en tidsgräns för hur länge de får användas.

Riktad reklam

En viktig anledning till medlemskap är möjligheten att generera riktad reklam som är anpassad efter vilka varor eller varukategorier som kunden brukar köpa, ålder, kön, adress, etc.

Personal

Kassasystem har normalt information om vem som satt i kassan vid en viss tidpunkt. Detta är absolut relevant, men behöver kompletteras med funktionalitet för att bli intressant i detta sammanhang. Annars blir det bara en klass till och ett fält i kvitto-klassen.

En möjlighet skulle vara att hålla reda på hur många varor varje medlem av kassapersonalen kan hantera per tidsenhet. En ofta kritiserad funktion i denna typ av system.

En annan skulle vara att införa någon typ av bonussystem som är beroende av hur mycket personen har sålt under månaden eller året.

Återköp

Ibland är inte kunder nöjda med ett köp utan vill göra ett återköp av hela eller delar av det. Detta påverkar många av de tidigare uppräknade funktionerna, till exempel lagerhållning, återbäring och poäng. Om varan ingått i en grupp av varor som kunden fått rabatt för måste detta också tas hänsyn till, samma sak o man har återbäring eller poängsystem.

Uppdelat köp

Det finns många sätt att betala på: kontanter, kort, Swish, poäng, bonuscheckar, presentkort, etc. Speciellt med de sistnämnda är det vanligt att dessa bara täcker delar av ett köp, och att kunden vill använda andra betalningsmedel för resten.

Parkera köp

Ett köp kan parkeras om kunden glömt något och vill gå tillbaka in i butiken eller hem och hämta plånboken. Detta är bara relevant om man faktiskt bygger själva kassan.

Loggning

Väldigt mycket av vad som sker i ett system av denna typ loggas, antingen för att det finns lagkrav på det, eller för att kunna få ut information om hur systemet används. Detta kan göras i databaser eller filer, som kan testas med hjälp av test doubles.

Rapporter

All den information som finns i systemet ger möjlighet att generera rapporter till olika delar av verksamheten. Dessa kan till exempel vara i html eller skrivas ut på skrivare. Alternativt kan man testa att generera diagram i html eller bildformat.

Kompilator eller interpretator

Denna uppgift är lämplig för grupper på tre personer. Detta eftersom uppgiften har en naturlig uppdelning i tre delar: lexer, parser och kodgenerator eller interpretator. Det är också önskvärt med tidigare erfarenhet av formella grammatiker, till exempel från AUTO eller PROP. En möjlig utökning om man är fyra skulle vara att implementera både en kodgenerator och en interpretator, eller istället göra ett interpreterande språk där man genererar ett enkelt mellanspråk som sen interpreteras istället för att interpretationen sker direkt på parse-trädet.

Språket som kompilatorn ska hantera bör vara enkelt, men inte alltför enkelt, då kommer kodgeneratorn att bli trivial. Det är lämpligt att gruppen tillsammans börjar med att ta fram en grammatik för språket så att alla är överens. Ett dynamiskt typat språk med heltal och textsträngar, input, output, tilldelning, addition, subtraktion och strängkonkatenering, if-satser, while-loopar samt eventuellt funktioner med parametrar och returvärden bör räcka långt. Om man är flera personer i gruppen går det att lägga till ytterligare språkliga mekanismer.

Tillståndsmaskiner är vanliga i samband med kompilatorer, andra testdesigntekniker är antagligen svårare att applicera om man inte har ett mer utvecklat språk där saker som typsystem och precedensregler skulle kunna leda till situationer där andra tekniker är mer användbara.

En profiler bör vara relativt lätt att testa på denna uppgift, bara koden som ska kompileras är tillräckligt stor. Om man gör både en kompilator och en interpretator skulle man till exempel kunna jämföra hur snabba dessa är och se om man kan optimera någonstans.

Lexern

Lexern, eller lexikalanalysatorns, uppgift är att översätta källkoden till en ström av lexikala element eller tokens där en token är ett objekt som innehåller en typ och ett värde. Medan den gör det plockar den samtidigt bort allt som inte är intressant för den fortsatta processen, som blanksteg, blankrader, kommentarer, etc. Till exempel skulle

```
if(age==34){ /* INGET ÄN */ }
```

i Java generera åtta tokens:

Typ:	IF	LPARAN	ID	EQUAL	INT	RPARAN	BEGIN	END
Värde:	if	(	age	==	34	)	{	}

Hur lexern gör detta varierar beroende på språkets komplexitet. I många fall är de enkla att skriva för hand genom att titta på nästa tecken, men det är också vanligt att en ändlig automat används.

Lexern kan få koden den ska tolka från olika källor: en sträng eller annan datakälla i minnet som en ström eller Reader, eller från en fil. Ofta har den möjlighet att använda bägge. Vid enhetstestning vill man antagligen undvika att använda riktiga filer i många fall, och dessa kan då simuleras med hjälp av test doubles.

Det finns också flera sätt att implementera strömmen av tokens. Ett enkelt sätt är att man har en metod som varje gång den anropas ger nästa token. Ett annat är att faktiskt implementera en strömklass.

Om lexern stöter på något den inte kan tolka i koden måste detta rapporteras till användaren. Det är dock inte lexerns uppgift att skriva ut det, den ska bara rapportera problemet. Detta kan göras

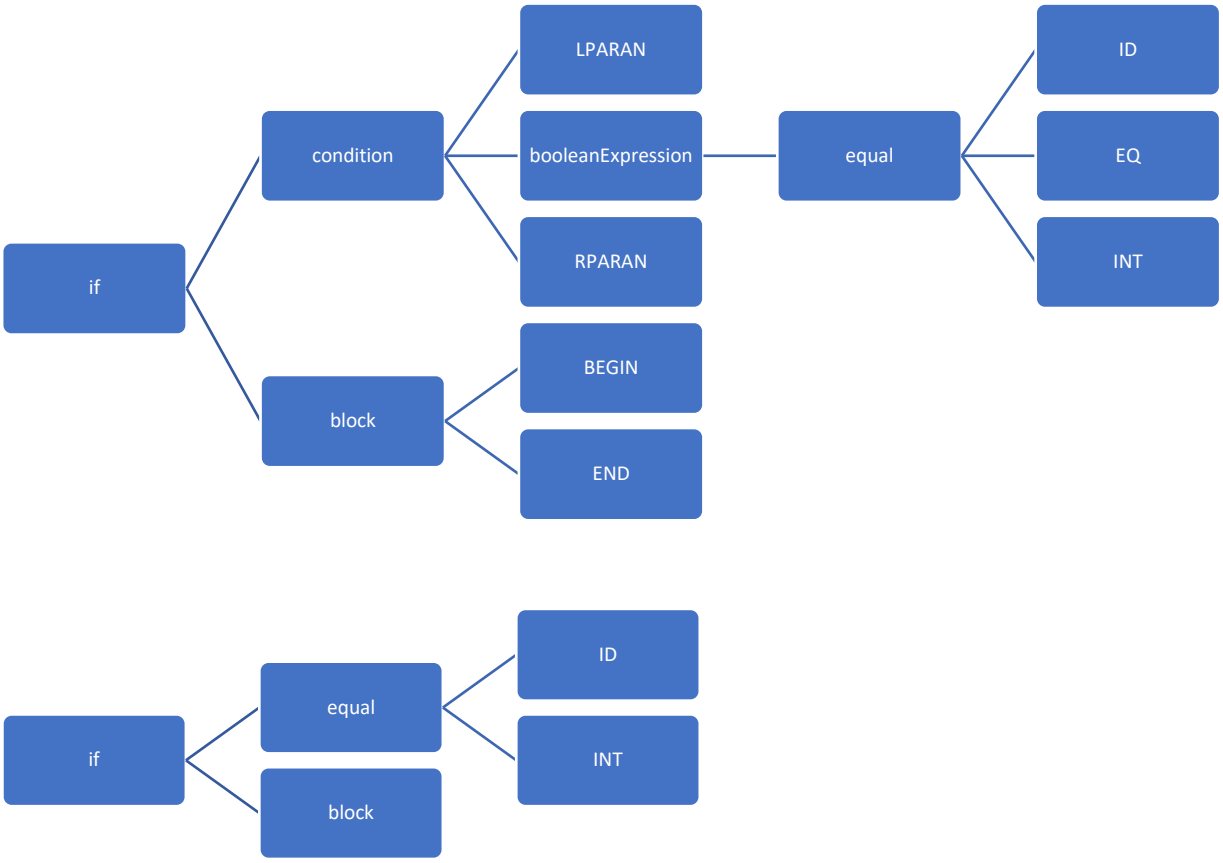
genom att kasta ett undantag, eller genom en Observer eller lyssnare. Det andra alternativet är antagligen bättre eftersom det tillåter lexern att fortsätta tolka resten av koden.

Parsern

Parserns uppgift är att ta strömmen av tokens och konvertera den till ett parse-träd som representerar strukturen hos koden. Detta parse-träd kan se olika ut beroende på språk och kompilatorns fördelning av funktionalitet mellan parsern och kodgeneratorn eller interpretatorn. De bägge träden nedan representerar bägge Java-exemplet ovan. Den första representerar kodens struktur väldigt exakt, den andra kodens logiska uppbyggnad. Allt som finns med i koden bara för att hjälpa kompilatorn är bortplockat.

Det finns flera standardvarianter för parsers som är kapabla att tolka olika komplexa språk. För det rekommenderade språket bör en så kallad recursive descent parser[10] vara fullt tillräcklig.

Om parserns jobb är att matcha inputen exakt, som i det första exemplet nedan, så är den antagligen lite för enkel för uppgiften. Det är bättre att lägga lite mer funktionalitet i parsern, så att den reducerar parseträdet, rapporterar fel (ej via utskrift), försöker återhämta sig om den inte kan matcha en token genom att kasta bort tokens tills den hittar en plats den kan fortsätta från, etc. Självklart går det också att hjälpa de andra deltagarna i gruppen, till exempel med att implementera en strömklass, eller att skriva själva programmet som binder samman de tre delarna.



[10] https://en.wikipedia.org/wiki/Recursive_descent_parser

Kodgenerator eller interpretator

Det sista steget är att faktiskt använda sig av parse-trädet. Detta genom att igenom parseträdet, till exempel med en Visitor, och antingen genererar kod (i en kompilator) eller utför det som ska ske (i en interpretator). I vissa språk är det tillräckligt att gå igenom trädet en gång, i andra måste man göra det flera gånger för att samla in information som behövs för de senare stegen.

Om ni väljer att implementera en kompilator är det antagligen bäst att generera kod i ett annat högnivåspråk, tex Java, istället för maskinkod eller bytekod om man inte väldigt gärna vill göra det ”på riktigt”. Det var så de första versionerna av C++ jobbade. De översatte C++-koden till C-kod som sen kompilerades till maskinkod av existerande C-kompilatorer.

Det kan också vara värt att implementera ett mallsystem för kodgenereringen så att detta inte behöver hårdkodas.

Precis som för lexern vill man troligtvis undvika att testa mot riktiga filer så mycket som möjligt, här bör det dock var lätt att undvika detta.

Alternativet är att istället skriva en interpretator som utför det som ska ske medan den går igenom parse-trädet. En sådan kommer att behöva datastrukturer för att hålla reda på saker som variabler och eventuella funktioner. För variablerna kommer den också att behöva skilja på olika datatyper.

Oavsett vilken version man väljer så kan saker gå fel även här, till exempel om en variabel inte är initierad innan den ska läsas av, eller om den är av fel typ. Här är det dock inte möjligt att fortsätta efter ett fel eftersom det inte finns något generellt sätt att hitta en återställningspunkt.

Ytterligare projektidéer

De resterande idéerna är inte lika utvecklad som de ovanstående. En grupp som är intresserad av någon av dem kommer därför att behöva ägna lite tid åt att sätta sig in i området och välja ut lämpliga uppgifter för deltagarna. I vissa fall krävs också tidigare kunskap inom området.

Diskutera gärna det planerade upplägget med kursledningen så snart som möjligt om ni avser att använda något av dessa idéer.

Bildmanipulering

Det finns ett stort antal algoritmer inom bildmanipulering[11] som skulle kunna implementeras, eventuellt i kombination med ett bibliotek för att läsa och/eller skriva bilder i något eller några bildformat. Ett sådant format behöver inte vara något redan existerande, utan kan vara ett som gruppen själv designar.

Det är svårt att bedöma denna uppgifts lämplighet eftersom det beror på vad man väljer att implementera och vilka förkunskaper man har. Att läsa på om allt som skulle krävas är inte görbart inom ramen för projektet, så denna uppgift bör bara väljas om gruppen har tidigare erfarenhet inom området.

Det är lätt gjort att algoritmimplementationer blir svårtestade annat än i sin helhet, så uppgiften är lämplig för någon som vill experimentera med testbarhet. Den är också utmärkt för profilers, och andra test av hastighet och effektivitet.

Om man implementerar att läsa och skriva ett bildformat till fil så är det ett bra ställe att testa på att använda test doubles eller mockobjekt.

Ekvivalensklassuppdelning bör var relativt enkelt att applicera, medan beslutstabeller antagligen blir svårt. Det enklaste stället att använda tillståndsmaskiner är antagligen tolkning av ett filformat, men det kan också vara relevant i vissa algoritmer.

Granskning av uppgiften kommer att ställa krav på deltagarna att ha åtminstone någon kunskap om algoritmerna för det som ska granskas, och det är inte nödvändigtvis så att det bästa sättet att granska är att läsa rad för rad från början till slut.

Grafbibliotek

Precis som för bilder finns det ett stort antal algoritmer som opererar på grafer[12] som skulle kunna implementeras: sökning, spännande träd, layout[13], etc. Ett plus jämfört med bildmanipulering är att ganska många som går kursen har erfarenhet av grafer från ALDA. Många andra ska gå ALDA i vår, vilket gör att tiden som går åt till att läsa in sig på området delvis kan fås tillbaka där.

Det är lätt gjort att algoritmimplementationer blir svårtestade annat än i sin helhet, så uppgiften är lämplig för någon som vill experimentera med testbarhet. Den är också utmärkt för profilers, och andra test av hastighet och effektivitet.

Ekvivalensklassuppdelning bör var relativt enkelt att applicera, medan beslutstabeller antagligen blir svårt. Tillståndsmaskiner kan eventuellt vara relevant i vissa algoritmer.

[11] https://en.wikipedia.org/wiki/Digital_image_processing
[12] https://en.wikipedia.org/wiki/Category:Graph_algorithms
[13] https://en.wikipedia.org/wiki/Graph_drawing

Lärplattform

En lärplattform som ilearn innehåller väldigt mycket funktionalitet, och skulle ensamt eller i kombination med ett studieadministrativt system som Daisy kunna utgöra en lämplig uppgift. Precis som för rouglike-biblioteket i det första förslaget skulle dock syftet inte vara att utveckla en ny version av ilearn eller Daisy, utan att implementera och testa ett klassbibliotek för delar av den bakomliggande modellen.

Samtliga testdesigntekniker bör vara lämpliga att använda på olika delar av ett sådant system. Beslutstabeller kan till exempel bestämma om en student är godkänd på ett moment, tillståndsmaskiner vilka delar av en kurs som är synlig, eller statusen i ett rättningsflöde. Ett tips är att inte låsa sig alltför mycket vid hur ilearn fungerar.

Beroende på exakt vad som implementeras skulle det vara möjligt att få in designmönster som visitor och decorator, mock-objekt (till exempel för en databas) och profilers på ett naturligt sätt. Det är också en uppgift där det skulle vara enkelt att bygga ett GUI för någon isolerad del om man vill testa på att använda ett verktyg för GUI-testning.

Samlarkortspel

Samlarkortspel[14] är en grupp av spel där två eller flera spelare tävlar mot varandra med hjälp av speciella kortlekar. Genren brukar räkna sitt ursprung till Wizards of the Coasts Magic: The Gathering[15] där deltagarna är trollkarlar som strider med magi och magiska varelser, men det finns många[16] varianter.

Just Magic har funnits i närmare 30 år, och har under tiden utvecklat ett stort och mycket komplext system med mängder av korttyper som ändrar på reglerna för spelet under spelet gång. Ett sådant system skulle fungera mycket väl som projektidé om man lyckas hitta en lämplig avgränsning. Det är inget ett krav att gruppen har tidigare erfarenhet av något existerande spel, men det hjälper. Det är lätt att fastna i att designa ett regelsystem annars.

Vilka testdesigntekniker som är lämpliga att använda beror helt på vilka mekanismer man väljer att implementera, men tillståndsmaskiner och beslutstabeller bör vara lätta att hitta användning för, och det är antagligen inte ett problem med ekvivalensklasser heller.

En profiler kommer antagligen inte att vara relevant eftersom det inte rör sig om något tidskritiskt eller stora mängder data.

Testramverk

I Test-Driven Development utvecklar Kent Beck ett enhetstestramverk i stil med JUnit med hjälp av testdriven utveckling. Det är en uppgift som är lätt att komma igång med, men som kan byggas ut praktiskt taget i oändlighet. I boken använder Beck testramverket för att testa sig självt. Det är inte ett krav.

[14] https://en.wikipedia.org/wiki/Collectible_card_game
[15] https://en.wikipedia.org/wiki/Magic:_The_Gathering
[16] https://en.wikipedia.org/wiki/List_of_collectible_card_games

Kalkylprogram

Ett kalkylprogram i stil med Excel skulle också kunna vara en möjlighet. Grunden är relativt enkel med ett rutnät med innehåll i cellerna. Detta kan sen byggas ut med olika funktioner där man också kan få in enkel parsning, generering av diagram till html eller en bild, etc.

Matcha tre

”Matcha tre-spel”[17] som Bejeweled och Candy Crush är en kategori av spel där spelaren ska gruppera ihop tre eller fler av samma sak. När detta görs plockas de antingen bort från spelplanen eller leder till någon annan effekt som att en ny sak bildas eller en rad med saker försvinner, eller något helt annat.

Grundfunktionerna i ett sådant spel är relativt enkla, och antagligen för litet för denna uppgift eftersom syftet inte är att utveckla ett spel utan att få något intressant att testa. Det går dock att lägga till många saker för att göra det mer intressant, till exempel:

- Matcha tre horisontellt, vertikalt och eventuellt diagonalt.
- Matcha fler.
- Matcha specifika mönster.
- Saker som måste matchas flera gånger innan de försvinner eller som påverkar sin omgivning.
- Slumpa nya saker när gamla försvinner.
- Maxantal drag.
- Vinstkriterier som ett visst antal matchade av vissa sorter, en totalpoäng, använda specifika powerups.
- Möjlighet att fortsätta efter att ha förlorat.
- Poäng och eller betyg på hur bra man klarat sig.
- Powerups och effekter.
- High score-lista.
- Speciella typer av saker.
- Uppdrag som sträcker sig över flera spelomgångar.
- Liv.
- Konto för användaren.
- Vänner man kan skicka och ta emot liv från.

[17] https://en.wikipedia.org/wiki/Tile-matching_video_game